2021

# Architecture

TWITTER BUT FOR CODE

FONTYS S65

## Table of Contents

## Document History

| *Version* | *Changes* | *Author* |
|---|---|---|
| *0* | Initial document layout | Arjen |
| *0.05* | First version of user stories, functionals and non-functionals | Arjen |
| *0.1* | Added tools and frameworks and first diagrams | Arjen |

## Introduction

This document will give an insight into the architecture of the “twitter but for code” application. The document will contain choices that I have made in terms of how we will be setting up the application. The document will also contain a few diagrams that will illustrate the structure of the application and the communication between the different components of the application.

## User stories

1. As a user I want to post my code snippets.
2. As a user I want to add friends to see their friend only posts.
3. As a user I want to be able to remove friends.
4. As a user I want to be able to like post.
5. As a user I want to be able to revision a post.
6. As a user I want to be able to report a post.
7. As a user I want to be able to comment on a post.
8. As a user I want to set who can see which posts.
9. As a user I want to remove a post.
10. As a user I want to edit a post.
11. As a user I want to be able to follow other users to see their newest posts.
12. As a user I want to be able to tag my post with the language it is written in.
13. As a guest I want to be able to look up public posts.
14. As a guest I want to be able to register an account.
15. As a guest I want to be able to login to an account.
16. As a admin I want the ability to edit or remove any user account or posts.
17. As a admin I want to have a feed where I can see reported post and handle them accordingly

## Non-functional requirements

1. **The storing and processing of user data should be along the guidelines of GDPR.**
   To handle user data along the guidelines of GDPR, we will have to make choices in who gets to see the user data and how much they get to see of the data. Users should for instance not be able to see other users data without permission.
2. **Users should be able to protect their own data.**
   There might be an occasion where the data of a user is meant to be private. Users should therefore be able to protect their own data when they want to and make it public later if they want to.
3. **It should be possible for a  user to request all their own data.**
   Data should be grouped by user, so that a user would be able to request all their own data. This also means that the user, if they for instance stopped doing research, can request for all their data to be removed from the system if they wish to do so.
4. **At least 50% of the written code needs to be tested.**
   To make sure that the code of the backend works properly in all thinkable scenarios and stays that way if changes are made to the code. All the code of the backend should be tested for at least 50% before it is pushed to the development branch.
5. **The application should be available 24/7.**
   The application will be used to save and get a lot of data. This will be done from a lot of different times, all over the world. Thus the application should be able to run at all times, so a user doesn't have to wait for a certain time for the application to be online.
6. **The application must be scalable.**
   To ease further development of the application, it is important that the application is scalable. This means that it should be easy to add new services and components to the system. It should also be able to replace components if needed.

## Tools and frameworks

- **React:** Front-end development.
- **Node:** Compiling and building react app.
- **Spring:** Back-end management.
- **Maven:** Compiling and building back-end project.
- **Intellij:** IDE for the backend code.
- **Visual Studio Code:** IDE for the frontend code.
- **Github:** Maintaining repository.
- **Gitkraken:** Interface for managing git repository.
- **Trello:** Keeping track us user stories of the project.
- **Jenkins:** Continuous integration and continuous deployment.
- **Kubernetes:** Resource management of the microservices.

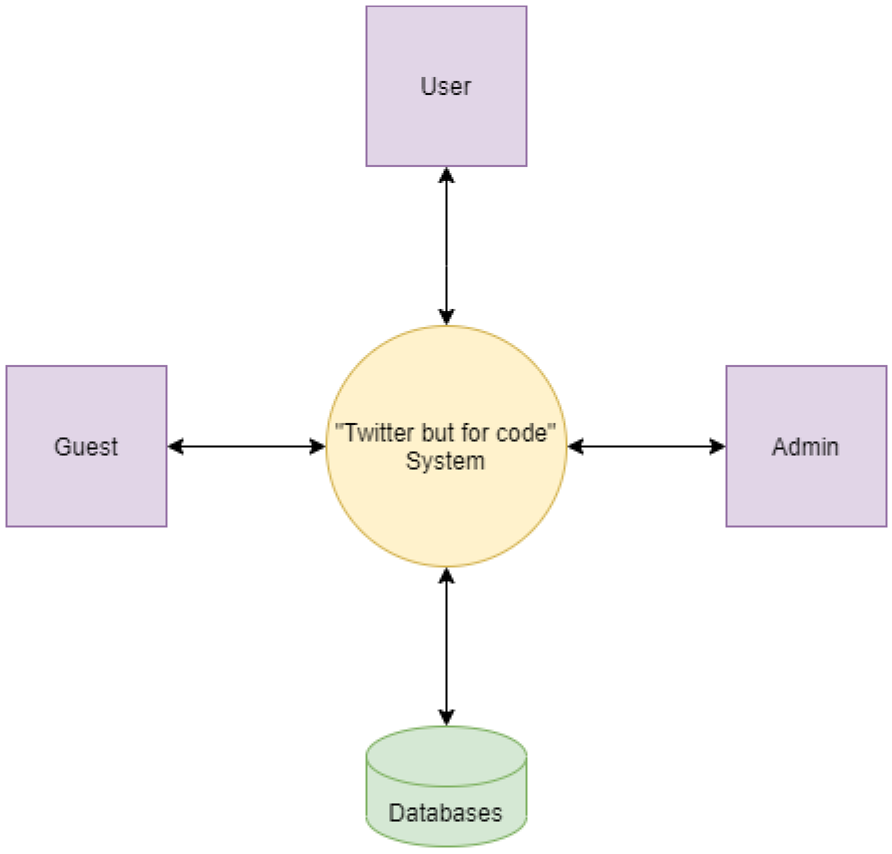
## System Context Diagram

In the system context diagram we show the system in the context of the rest of the world. This means we show the different users and systems that interact with our system.

The guests have no more access then the ability to look at public profiles and posts and to create a user account.

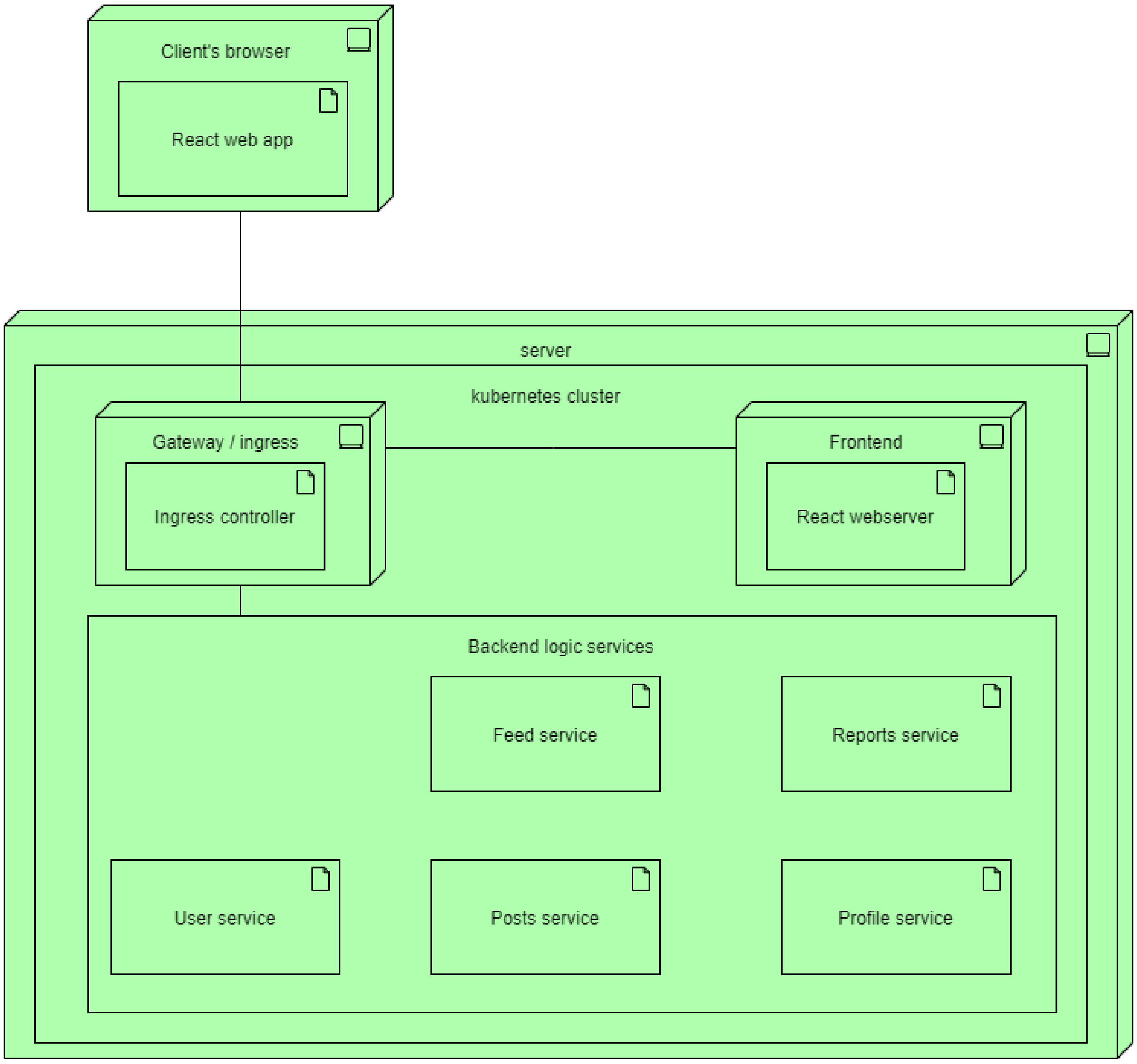
The users can create their own posts, manage their account, manage their friends, follow users and react to other posts. They also have a recent feed where the can look at the most recent new posts of friends and users they follow.

An admin can edit and remove any post or user. They also can look at a reported feed where they can see the reported posts and mark these as resolved after handling a report.



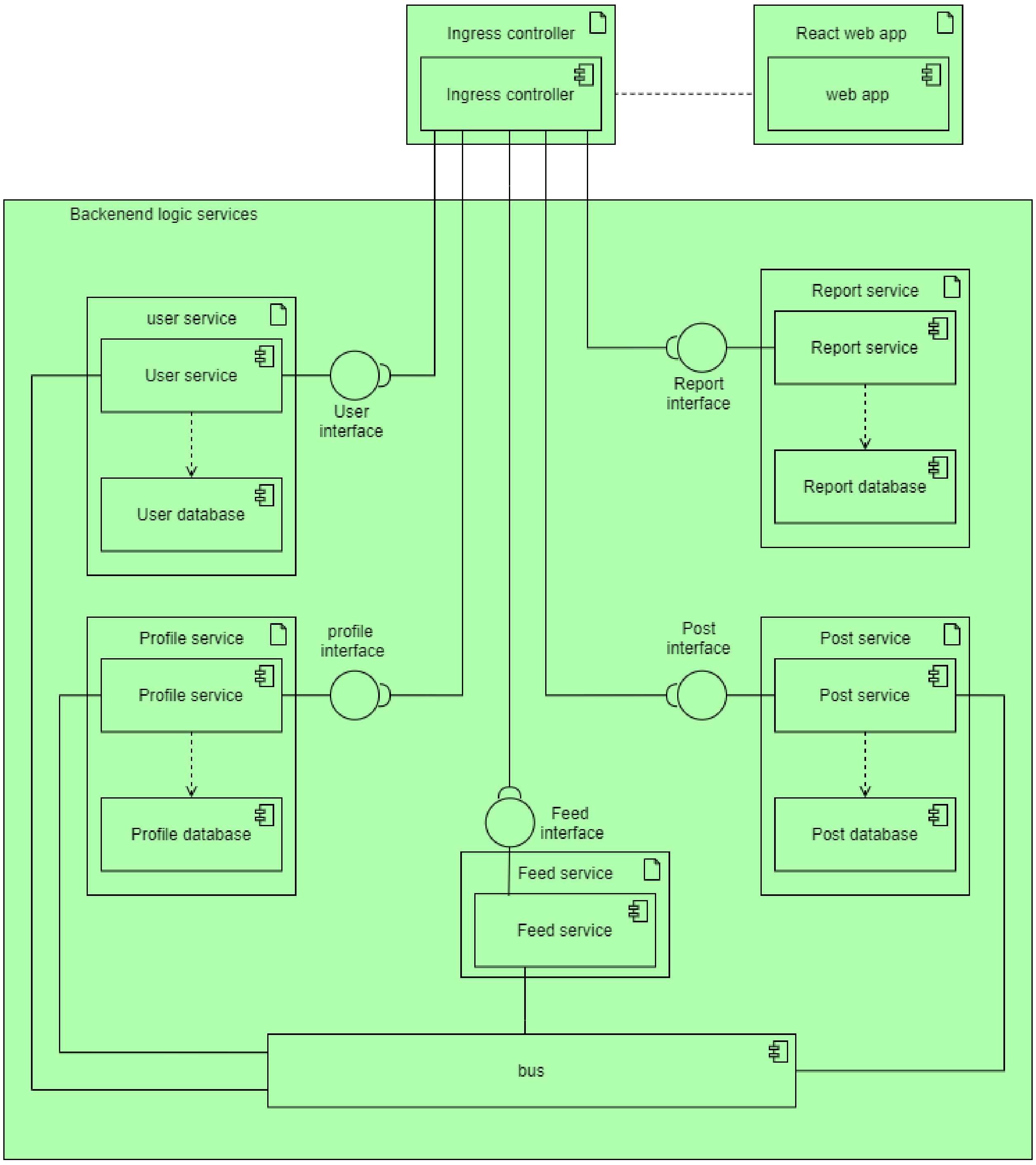
## Container Diagram

In the container diagram we show all the parts of the system. Everything will be located in a Kubernetes cluster for scalability. We use a Gateway to handle communication between the cluster client and the backend services.



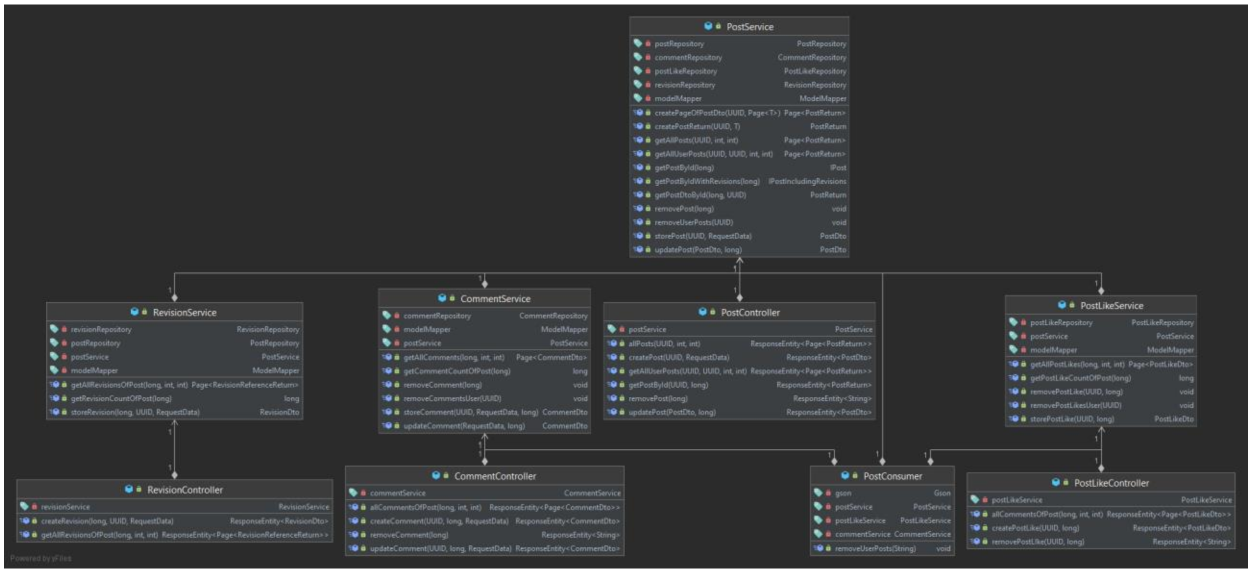
## Component Diagram

In the component diagram we show all the components of the system. All services that have a database have this shown in the diagram. The internal communication between components will be handled by a bus, while the external communication will go through a gateway. The services expose an interface for the client to communicate with.

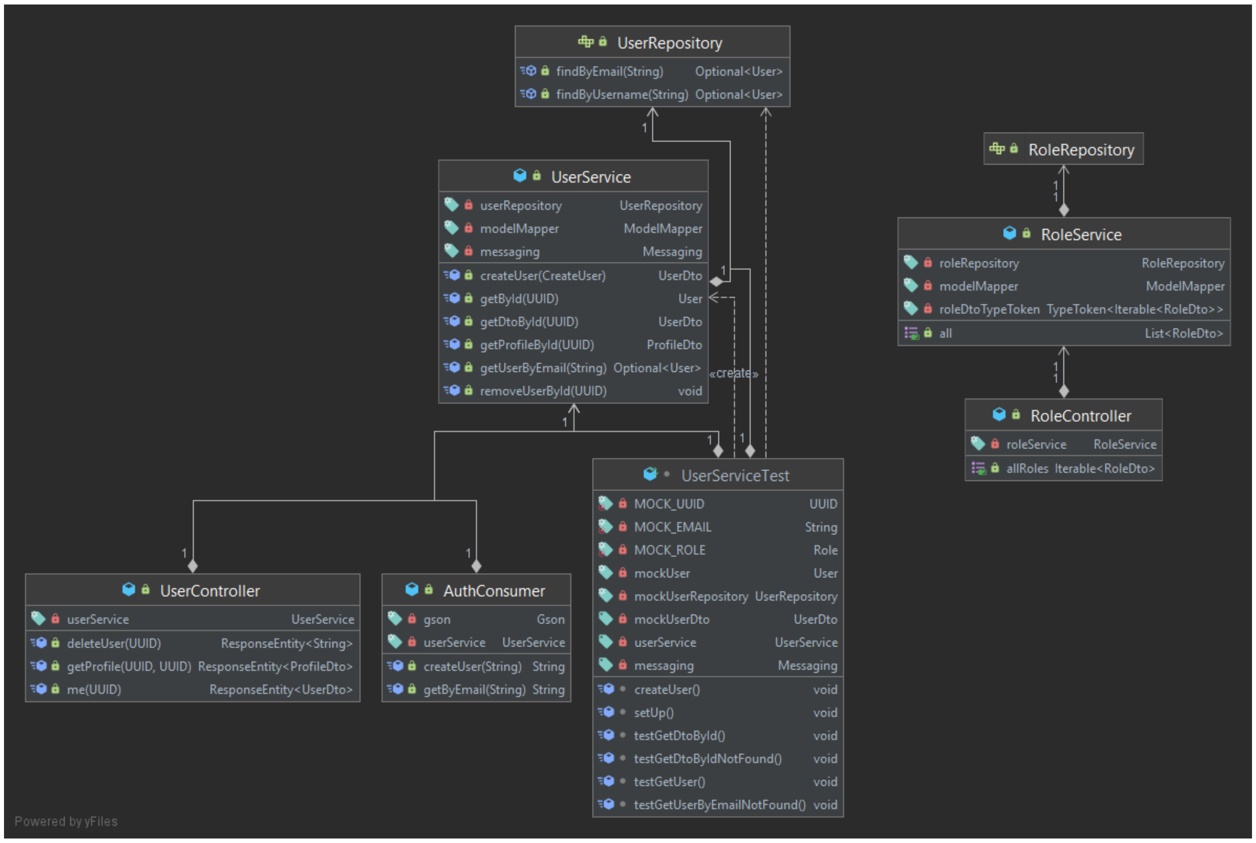


# Class Diagram

## Post service

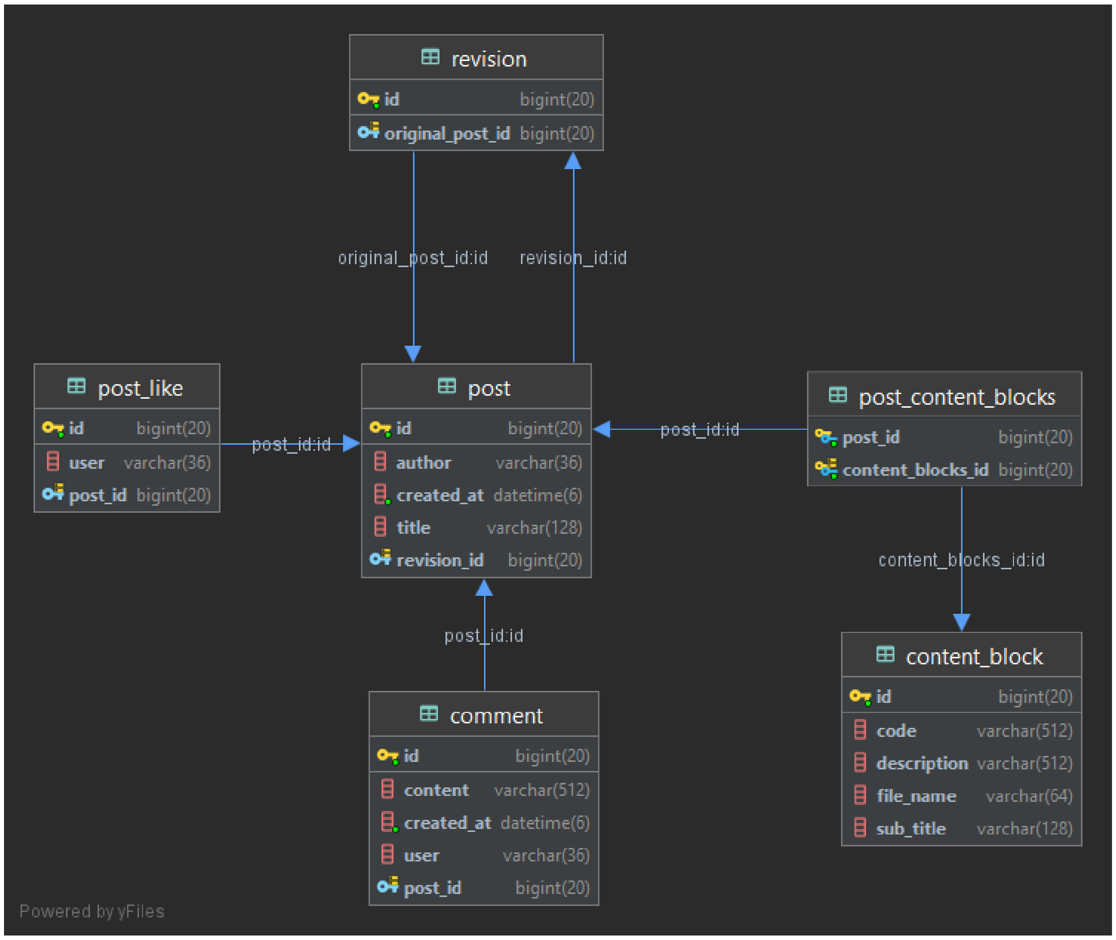

PostService
postRepository PostRepository
commentRepository CommentRepository
postLikeRepository PostLikeRepository
revisionRepository RevisionRepository
modelMapper ModelMapper
createPageOfPostDto(UUID, Page<T>) Page<PostReturn>
createPostReturn(UUID, T) PostReturn
getAllPosts(UUID, int, int) Page<PostReturn>
getAllUserPosts(UUID, UUID, int, int) Page<PostReturn>
getPostById(long) IPost
getPostByIdWithRevisions(long) IPostIncludingRevisions
getPostDtoById(long, UUID) PostReturn
removePost(long) void
removeUserPosts(UUID) void
storePost(UUID, RequestData) PostDto
updatePost(PostDto, long) PostDto
RevisionService
revisionRepository RevisionRepository
postRepository PostRepository
postService PostService
modelMapper ModelMapper
getAllRevisionsOfPost(long, int, int) Page<RevisionReferenceReturn>
getRevisionCountOfPost(long) long
storeRevision(long, UUID, RequestData) RevisionDto
CommentService
commentRepository CommentRepository
modelMapper ModelMapper
postService PostService
getAllComments(long, int, int) Page<CommentDto>
getCommentCountOfPost(long) long
removeComment(long) void
removeCommentsUser(UUID) void
storeComment(UUID, RequestData, long) CommentDto
updateComment(RequestData, long) CommentDto
PostController
postService PostService
allPosts(UUID, int, int) ResponseEntity<Page<PostReturn>>
createPost(UUID, RequestData) ResponseEntity<PostDto>
getAllUserPosts(UUID, UUID, int, int) ResponseEntity<Page<PostReturn>>
getPostById(UUID, long) ResponseEntity<PostReturn>
removePost(long) ResponseEntity<String>
updatePost(PostDto, long) ResponseEntity<PostDto>
PostLikeService
postLikeRepository PostLikeRepository
postService PostService
modelMapper ModelMapper
getAllPostLikes(long, int, int) Page<PostLikeDto>
getPostLikeCountOfPost(long) long
removePostLike(UUID, long) void
removePostLikesUser(UUID) void
storePostLike(UUID, long) PostLikeDto
RevisionController
revisionService RevisionService
createRevision(long, UUID, RequestData) ResponseEntity<RevisionDto>
getAllRevisionsOfPost(long, int, int) ResponseEntity<Page<RevisionReferenceReturn>>
CommentController
commentService CommentService
allCommentsOfPost(long, int, int) ResponseEntity<Page<CommentDto>>
createComment(UUID, long, RequestData) ResponseEntity<CommentDto>
removeComment(long) ResponseEntity<String>
updateComment(UUID, long, RequestData) ResponseEntity<CommentDto>
PostConsumer
gson Gson
postService PostService
postLikeService PostLikeService
commentService CommentService
removeUserPosts(String) void
PostLikeController
postLikeService PostLikeService
allCommentsOfPost(long, int, int) ResponseEntity<Page<PostLikeDto>>
createPostLike(UUID, long) ResponseEntity<PostLikeDto>
removePostLike(UUID, long) ResponseEntity<String>
1
Powered by yFiles

## User service

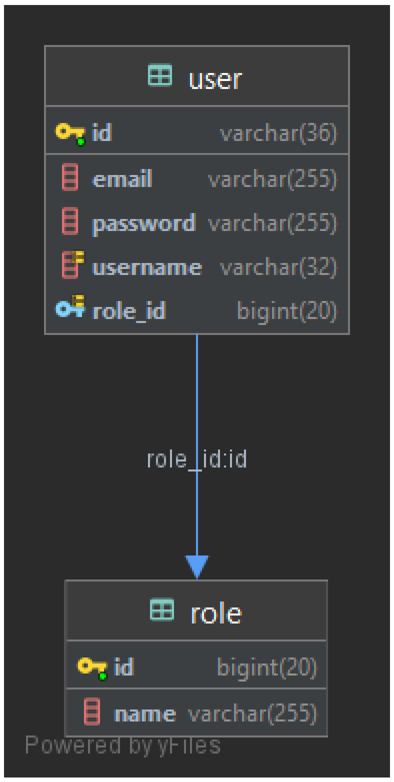

UserRepository
findByEmail(String) Optional<User>
findByUsername(String) Optional<User>
UserService
userRepository UserRepository
modelMapper ModelMapper
messaging Messaging
createUser(CreateUser) UserDto
getById(UUID) User
getDtoById(UUID) UserDto
getProfileById(UUID) ProfileDto
getUserByEmail(String) Optional<User>
removeUserById(UUID) void
«create»
RoleRepository
RoleService
roleRepository RoleRepository
modelMapper ModelMapper
roleDtoTypeToken TypeToken<Iterable<RoleDto>>
all List<RoleDto>
RoleController
roleService RoleService
allRoles Iterable<RoleDto>
UserServiceTest
MOCK_UUID UUID
MOCK_EMAIL String
MOCK_ROLE Role
mockUser User
mockUserRepository UserRepository
mockUserDto UserDto
userService UserService
messaging Messaging
createUser() void
setUp() void
testGetDtoById() void
testGetDtoByIdNotFound() void
testGetUser() void
testGetUserByEmailNotFound() void
UserController
userService UserService
deleteUser(UUID) ResponseEntity<String>
getProfile(UUID, UUID) ResponseEntity<ProfileDto>
me(UUID) ResponseEntity<UserDto>
AuthConsumer
gson Gson
userService UserService
createUser(String) String
getByEmail(String) String
1
Powered by yFiles

## Database Diagram

### Post service

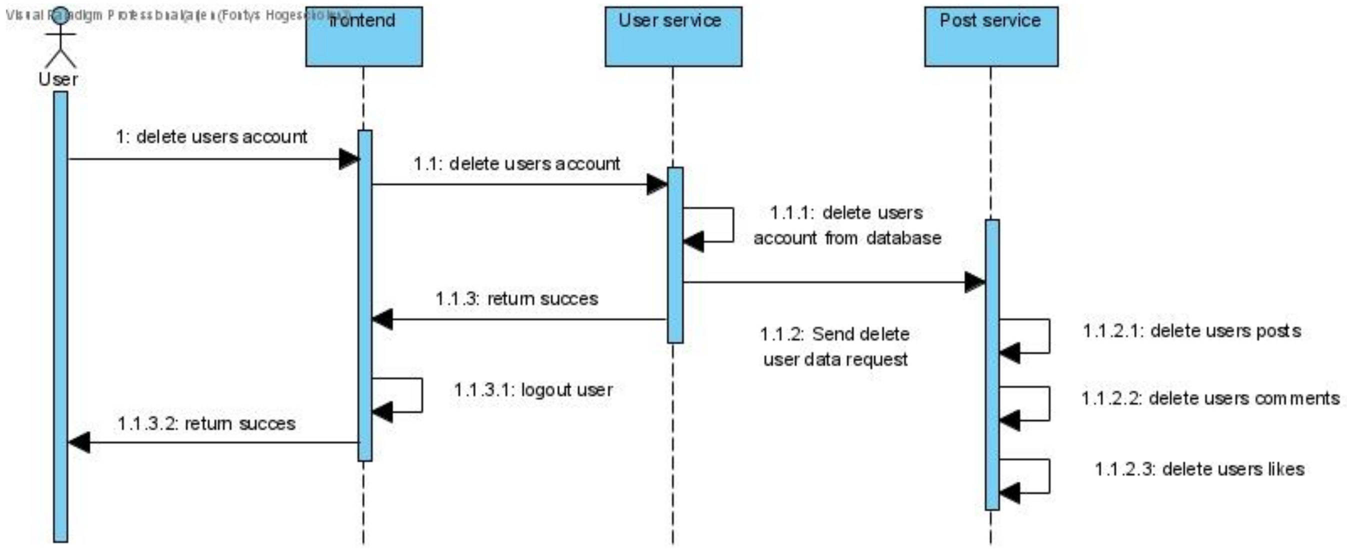


## User service



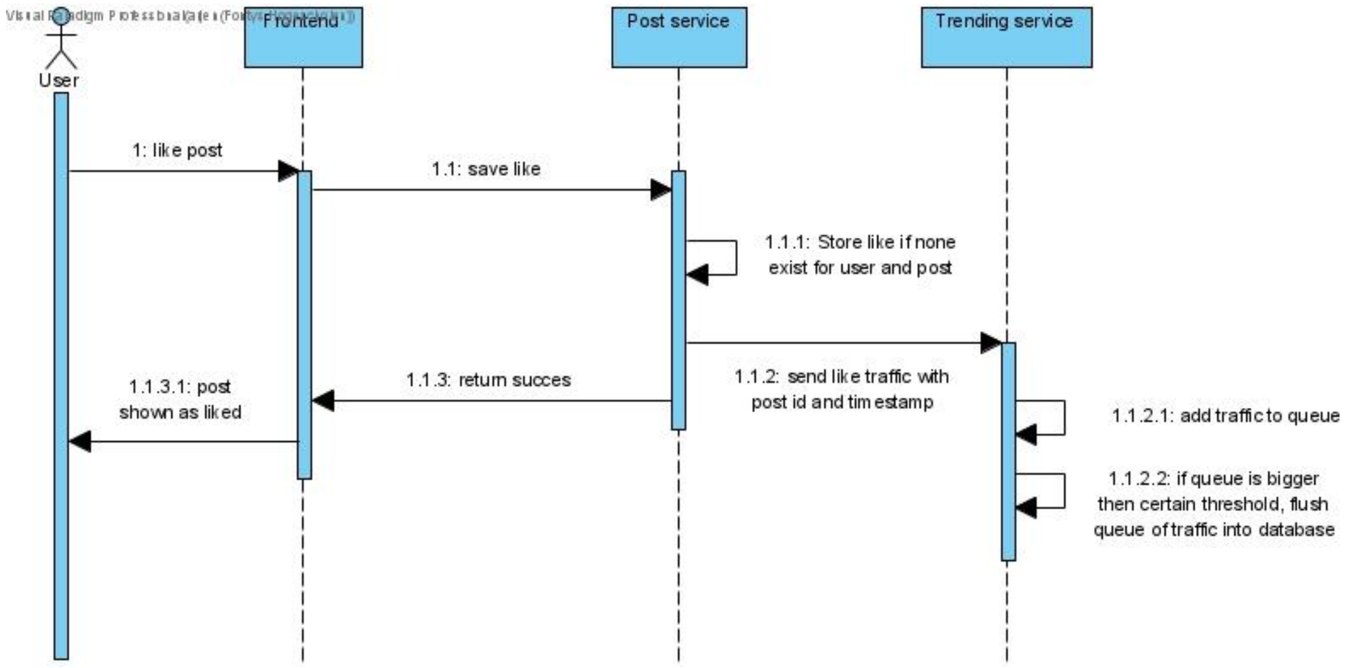
## Sequence Diagram

### Remove users account


Visual Paradigm Professional (Fontys Hogeschool)
User
Frontend
User service
Post service
1: delete users account
1.1: delete users account
1.1.1: delete users account from database
1.1.2: Send delete user data request
1.1.2.1: delete users posts
1.1.2.2: delete users comments
1.1.2.3: delete users likes
1.1.3: return succes
1.1.3.1: logout user
1.1.3.2: return succes

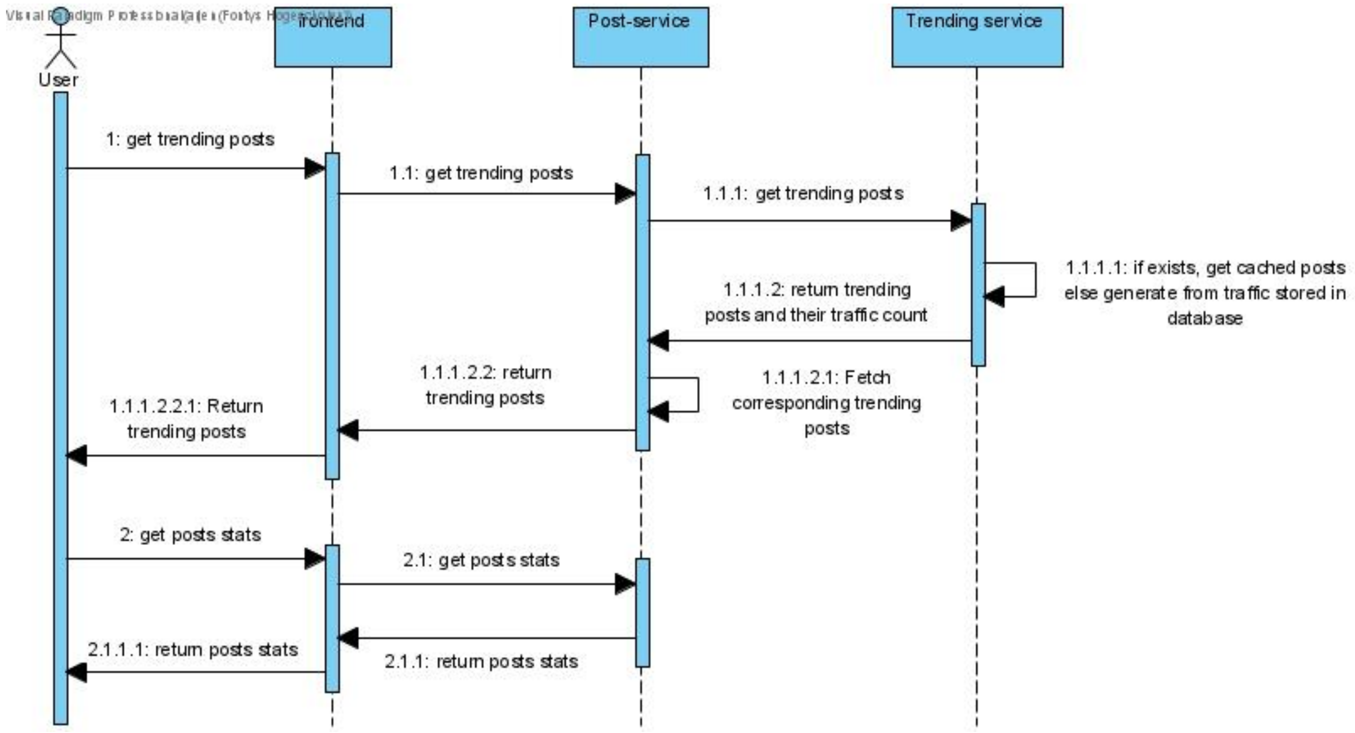
### Liking a post

*Same pattern if a user comments or creates a revision*


Visual Paradigm Professional (Fontys Hogeschool)
User
Frontend
Post service
Trending service
1: like post
1.1: save like
1.1.1: Store like if none exist for user and post
1.1.2: send like traffic with post id and timestamp
1.1.2.1: add traffic to queue
1.1.2.2: if queue is bigger then certain threshold, flush queue of traffic into database
1.1.3: return succes
1.1.3.1: post shown as liked

## Getting trending posts

*This is the same for all different time frames of trending posts*


Visual Paradigm Professional (Fontys Hogeschool)
User
frontend
Post-service
Trending service
1: get trending posts
1.1: get trending posts
1.1.1: get trending posts
1.1.1.1: if exists, get cached posts else generate from traffic stored in database
1.1.1.2: return trending posts and their traffic count
1.1.1.2.1: Fetch corresponding trending posts
1.1.1.2.2: return trending posts
1.1.1.2.2.1: Return trending posts
2: get posts stats
2.1: get posts stats
2.1.1: return posts stats
2.1.1.1: return posts stats

# Interface Declaration

## RabbitMQ messages

The messages can be either be send without expecting a return or be send expecting a return. There is a standardized class for this that can be used to send messages. This message class has a method for both only sending a message and sending and returning a message. These methods need to be given a queue (String) and the object containing the data that you want to send (any object). If it is the send and receive method a class type of the object you expect back (any object) should also be provided.

If a message is send it will be wrapped in a request object that will contain the data object that you want to send. This is done so it is more easily converted back into an object once it is received by a consumer. A consumer can then send back a response object. This response object is almost identical to the request object, except that it contains a Boolean for if the request was a success and a possible exception if it was not. This ensures that if there is an exception, this exception will be send back to the sender and RabbitMQ will consider the request done and won't re add the request to the queue creating an infinite loop.

## Auth and user service

**Queue:** register-user
**Sender:** Auth service
**Receiver:** User service
**Description:** The auth service sends a request to create a user in the user database. The password of this user is already hashed by the auth service.
**Data send:**

- **CreateUser:**
  - **String** username
  - **String** email
  - **String** password
  - **RoleDto** role:
    - **long** id
    - **String** name

**Data received:**

- **UserDto:**
  - **UUID** id
  - **String** username
  - **RoleDto** role
  - **String** email

**Queue:** get-user-by-email
**Sender:** Auth service
**Receiver:** User service
**Description:** The auth service sends a request to find and return a user by their email.
**Data send:**

**String** email

**Data received:**

**User:**

**UUID** id
**String** username
**String** email
**String** password
**Role** role:

**long** id
**String** name

## Post and user service

**Queue:** remove-data-user
**Sender:** User service
**Receiver:** Post service
**Description:** The user service sends a request to the post service to delete all the users data. This includes likes, posts and comments.
**Data send:**

**RemoveUserDto:**

**UUID** id**;**

## Post and trending service

**Queue:** post-like-traffic
**Sender:** Post service
**Receiver:** Trending service
**Description:** The post service sends a message to the trending service that there was a like on a specific post at a certain time
**Data send:**

**RabbitTrafficDto**

**long** postId;
String dateTimeString;

**Queue:** post-comment-traffic
**Sender:** Post service
**Receiver:** Trending service
**Description:** The post service sends a message to the trending service that there was comment on a specific post at a certain time
**Data send:**

> **RabbitTrafficDto**
>> **long** postId;
>> String dateTimeString;

**Queue:** post-revision-traffic
**Sender:** Post service
**Receiver:** Trending service
**Description:** The post service sends a message to the trending service that there was a revision on a specific post at a certain time
**Data send:**

> **RabbitTrafficDto**
>> **long** postId;
>> String dateTimeString;

**Queue:** trending-post-day
**Sender:** Post service
**Receiver:** Trending service
**Description:** The post service sends a request containing a page number and a page size to get the current trending posts of the last day. The trending service will then return the trending postId's in order with their respective traffic amount. It will also contain to total amount of pages, page size and current page.
**Data send:**

> **TrendingPostsRequest**
>> **int** pageNr;
>> **int** pageSize;

**Data received:**

> **TrendingPostPageDto**
>> **private List<TrendingPostDto>** trendingPosts;
>>> **long** postId;
>>> **long** counted;
>>
>> **int** maxPages;
>> **int** pageNumber;
>> **int** pageSize;

**Queue:** trending-post-week
**Sender:** Post service
**Receiver:** Trending service
**Description:** The post service sends a request containing a page number and a page size to get the current trending posts of the last week. The trending service will then return the trending postId's in order with their respective traffic amount. It will also contain to total amount of pages, page size and current page.
**Data send:**

> **TrendingPostsRequest**
>> **int** pageNr;
>> **int** pageSize;

**Data received:**

> **TrendingPostPageDto**
>> **private List<TrendingPostDto>** trendingPosts;
>>> **long** postId;
>>> **long** counted;
>>
>> **int** maxPages;
>> **int** pageNumber;
>> **int** pageSize;

**Queue:** trending-post-month
**Sender:** Post service
**Receiver:** Trending service
**Description:** The post service sends a request containing a page number and a page size to get the current trending posts of the last month. The trending service will then return the trending postId's in order with their respective traffic amount. It will also contain to total amount of pages, page size and current page.
**Data send:**

> **TrendingPostsRequest**
>> **int** pageNr;
>> **int** pageSize;

**Data received:**

> **TrendingPostPageDto**
>> **private List<TrendingPostDto>** trendingPosts;
>>> **long** postId;
>>> **long** counted;
>>
>> **int** maxPages;
>> **int** pageNumber;
>> **int** pageSize;

**Queue:** trending-post-year
**Sender:** Post service
**Receiver:** Trending service
**Description:** The post service sends a request containing a page number and a page size to get the current trending posts of the last year. The trending service will then return the trending postId's in order with their respective traffic amount. It will also contain to total amount of pages, page size and current page.
**Data send:**

```
TrendingPostsRequest
        int pageNr;
        int pageSize;
```

**Data received:**

```
TrendingPostPageDto
        private List<TrendingPostDto> trendingPosts;
                long postId;
                long counted;
        int maxPages;
        int pageNumber;
        int pageSize;
```